Bachelor's thesis

Degree programme in Information Technology

Networks

2016

Roman Zaynetdinov

DATA REPRESENTATION FOR THE RULES EDITOR

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR´S THESIS | ABSTRACT
TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme in Information Technology | Networks

February 2016 | 27

Petri Pellinen (Lekane Oy), Patric Granholm

Roman Zaynetdinov

DATA REPRESENTATION FOR THE RULES EDITOR

One way to have a reliable and solid software is to have a suitable data model. A well-chosen model helps to eliminate multiple issues from understanding the data to using the model properly in the software. As a result, such a model can lead to a simpler software implementation. When the software component is dependent on the data model, it is advisable to create a data model prior to starting the software implementation.

This thesis deals with creating a data model for the rules editor which will be included into the existing software of the company which commissioned this thesis. The data model should be reliable and transferred between the server side and the client side (client browser).

The thesis is structured as follows. Firstly, it describes the new software component to be built and its background. Secondly, it discusses t the limitation(restrictions) of the current system of the commissioning company. Then, it concludes with the most suitable data model which suits the current stack well without introducing new complexities and complicated changes to the company's system. Finally, it discusses alternative methods of solving the problem and evaluates if the problem has been solved correctly.

KEYWORDS:

Data model, Software, Databases, Data representation

CONTENTS

FIGURES

LIST OF TERMS AND ABBREVIATIONS

JEXL	Java Expression Language
JVM	Java Virtual Machine
JSON	JavaScript Object Notation
SQL	Structured Query Language
MySQL	Open Source Relational Database Management System
XML	Extensible Markup Language
URL	Uniform Resource Locator
RDBMS	Relational Database Management System
ORDBMS	Object-relational Database Management System
W3C	World Wide Web Consortium
REST	Representational State Transfer
enum	Enumerated Type

1 INTRODUCTION

Lekane is a technology company based in Espoo, Finland. Among other services, Lekane provides a Lekane Dialogue solution for its clients. The Dialogue allows a client company to analyze their users' behaviour on their website. The Dialogue also provides online chat and callback solutions for the client company website. Online chat can be used by a site visitor to receive an instant reply to their question. The callback option can be used by a site visitor to leave their phone number to the company agent, so that the site visitor could be reached immediately after.

To benefit from these features the most, a client company specifies a list of goals to control the appearance times of the chat and the callback for the targeted visitors. In this way the chat and the callback are visible on the website at the hours defined by the client. However, when a goal is triggered, it can react in different ways. It can either show chat, or callback offer to the site user, or not respond at all. Goal responses are controlled by complex logic and this logic layer is out of this thesis scope.

Each goal has multiple timespans, contexts, and rules. All of them control goal behaviour and determine when the goal should be triggered.

Timespan defines when the goal is in service. It describes goal availability times and dates based on agents operating hours.

As for the context, it defines a client website section. Therefore, the site could be split into multiple areas, such as shop area, news area, and so on. Each of those areas can have different behaviours.

In its turn, a rule defines a goal condition. A specified condition should succeed so that the goal can be triggered. Each condition is evaluated into a boolean expression. While a goal can have multiple rules, the client company can specify how they want these rules to function in practice. Either any rule can trigger the goal, or all rules must succeed to trigger the goal.

It needs to be mentioned here that current implementation of the goal specification editor does not exist yet. So the goals are manually edited in the property files by the company employees. This means that when a client wants to make even a slight change in the goal specification, they should notify Lekane by email. The present situation raises a question about the necessity of developing a rules editor. The main aim of the rules editor is to simplify the previously described request process and provide a self service solution for the clients.

While the default goal specification will remain in the property files, the rules editor will allow clients to replace goal rules provided by Lekane by applying their own rules to the goal.

Having analyzed the problem and having considered the possible solutions, a conclusion was made that rules editor will be a front end application. It will require three data models:

1. A data model for the actual editor

Rules should be somehow presented in the editor. This model should be flexible enough for the future expanding and most importantly reliable. This model should also allow dependencies between selected options. When one option is dependant on another option, we should be able to filter the current options by the selected parent value.

2. A data modelfor the transmission step

When a client saves the rules, they should be transferred to the server side. This model should be clear to understand. Besides, both server and browser should support it natively or have robust libraries for dealing with the data.

3. A data model for the server

It is worth noting that transmission process can introduce potential vulnerabilities to the data, thus when the server receives the rules, it should complete the following tasks: first, authenticate the user; second, verify that the user has the

privileges to modify the goal rules; finally, the server should validate the received list of goals.

1.1 Issues with existing solutions

Having researched the materials related to the topic, it can be stated that there are multiple existing solutions that are able to solve the problem being solved. All of these solutions provide methods for manipulating business rules. Some of them even provide the complete Business Rules Management System. The main solutions related to the problem mentioned above are the following:

1. Drools (Red Hat 2016) provide a complete Business Rules Management System. It provides multiple features and is highly customizable. The drawback is that this product is a standalone solution, thus it will be hard to integrate Drools into a company's software. Moreover, the solution provided by Drools seems to be excessive for the problem being solved.

2. ServiceNow (Service Now 2015) also provides a Business Rules Management System as one of their products. It has the same drawbacks that were listed above in relation to Drools. In addition this option is not free and is proprietary.

3. A Business Rules library (Powers 2013) could be a good candidate for the frontend side of the rules editor. However, it can be said that it is unpopular and has not been maintained for a couple of years.

Overall, it can be concluded that the problem being solved for the rules editor is simple, specific for company's back end, and not strictly related to the business rules.

2 METHODS

Having analyzed the problem and the scope of the present work for its possible solutions, it is crucial to examine the research and problem-solving methods and stages.

The first step of the research work is to identify which rule conditions clients would like to modify. Currently, there are several available criteria, but only part of them are being used. While some of the criteria are too complex for the end user, other criteria are irrelevant. A conclusion can be made that only the most used conditions should become available for the editing.

The second step is to remove the complexity of editing rules. Rules are now specified using JEXL and during the server execution, they are mapped to the Java methods. On the one hand, JEXL (Apache Commons JEXL) is an excellent choice to specify a condition in the rules editor. It perfectly provides all the necessary expression flexibility and extendability. On the other hand, it appears to be not user-friendly and it requires a learning curve. There should be a higher model representation in addition to the expression language.

If the front end and transmission data models do not have any boundaries, the rules editor will be created from scratch based on the data model proposed. The server model does have limitations. Therefore, the data model should be compatible with existing implementation.

The server side data model requires current server analysis to determine how rules are stored in the application as well as in the database.

Finally, the transmission phase allows to use various data formats such as XML, JSON and string representation.

2.1 What is XML?

XML is a markup language defining document rules (Rosenberg 2007). This format is widely used for cross application communications especially on the

Internet. Mulberry Technologies Inc. in their material on How and Why Are Companies Using XML identified the following benefits of XML (Usdin 2006):

Simplicity

XML documents are easily read by human and processed by the machine.

Openness

XML specification is defined by W3C as of XML 1.0 and is a free open standard.

Extensibility

Users are allowed to create as many new tags as they need and are not bound to the existing definitions.

Self-description

Any XML tag can contain an unlimited number of attributes. These attributes define additional element meta data, such as author name or version.

Separates content from presentation

XML tags define only the information of the document, allowing changing the document presentation without modifying the content.

Supports multilingual documents and Unicode

This is an important feature which supports multi international applications.

Facilitates the comparison and aggregation of data

XML documents have a tree structure, thus allowing efficient document comparisons and embedding other data types.

Rapid adoption by industry

XML documents have a wide support across the industry.

2.2 Advantages of JSON over XML

Following the logic of the research methods mentioned earlier in the present chapter, it seems relevant to discuss the advantages of JSON over XML.

JSON is an open document format defined by Douglas Crockford (Bray 2014). JSON as well as XML are both open web standards defined by RFC 7159 and RFC 4825 accordingly. Some of the JSON benefits over XML were identified by Tom Strassner from Tufts University in his research paper (Strassner n.d):

Fewer verbosity

JSON requires fewer characters to describe the document. Instead of opening and closing tags as XML, JSON uses braces (“{“, “}) and brackets (“[“, “]”), thus decreasing the total document file size.

Faster processing

Due to their smaller file size, JSON documents can be transferred and processed by the application faster.

Fewer resource usage

JSON processing uses less total CPU resources compared to XML.

Is a subset of JavaScript

JSON is a subset of JavaScript allowing to parse JSON natively in JavaScript applications.

2.3 System diagram

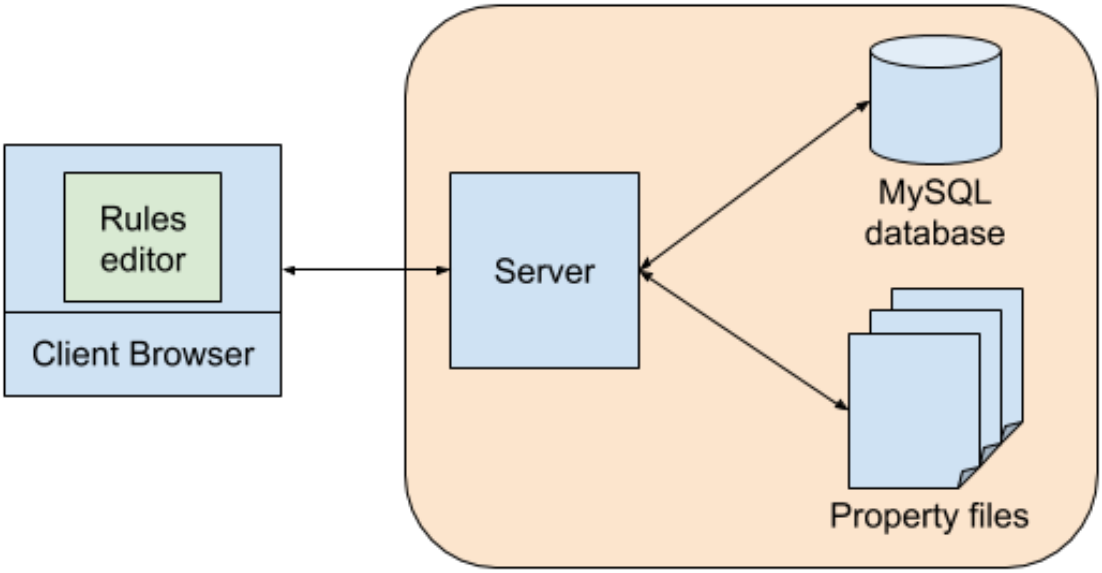

Figure 1. Rules editor workflow

Figure 1 helps to understand how the data will be communicating inside the current stack. The rules editor can communicate only with server. The server is able to access goals definitions in property files and load persisted goals from the database.

2.4 Client-to-Server communication

To provide an end point for the client browser to communicate with the server side application, a RESTful service was created on the server side. REST is the architecture style for creating web services. Typically, it provides a set of methods: GET, POST, PUT, DELETE for accessing, creating, modifying and deleting objects accordingly (Fowler 2010). The RESTful service is a service that implements all or some of the methods described earlier. REST was chosen for its simplicity, wide adoption by the community, and for being easy to create and maintain.

The server provides three endpoints allowing all necessary data manipulation. The endpoints are the following:

- GET: fetches goal rules from the database;
- POST: saves new set of rules to the database (overrides the previously saved rules, thus performing create operation);
- DELETE: removes all saved goal rules from the database.

It is worth noting that all these data manipulations are specific for each user. Users should be authenticated to the Lekane admin panel where the final rules editor will be located. Users should also be allowed to edit only the goals they have an access to.

The current admin panel was implemented using Apache Wicket. Wicket is a server side web framework for creating web applications. Currently it is being maintained by Apache Foundation. The Rules editor web application will be integrated inside the Wicket application, thus allowing us to reuse current authentication mechanisms. Wicket uses session-based authentication backed by cookies. The REST endpoints validate not only that the user is logged in, but also validate that the user is allowed to modify the goal rules.

In this way the complete communication process between the client browser and the server was analyzed.

2.5 Persistence

Passing to the question of the persistence, it is worth mentioning that goals originally are stored in the property files. When the server starts, it updates them in the database and loads them in the memory. If a goal has been modified before and these modifications are stored inside the database, these modifications will override the behaviour specified in the property files. There are three different goal representations throughout the application.

```
TrackingGoal.MyGoal.contexts=Web_Store_Context
TrackingGoal.MyGoal.rules=Two_Page_Views_Rule
…
TrackingContext.Web_Store_Context.regexp=.*/store/.*
…
TrackingRule.Two_Page_Views_Rule=PageViews > 1
```

Figure 2. Basic goal definition

In the property files, the goals are stored in the text format. Figure 2 shows a simple goal definition including defining a context and a rule for the goal. Users can freely modify, remove the existing goal definitions and they can create new ones. Goal rules expressions can also be modified freely. When the server starts, it parses the property files, thus forcing property files to have a specific formatting and goals to have a precise definition. When all the files have been read, previously saved changes from the database are being applied to the goals data. All this persistence is implemented by the Java Hibernate library.

The final goal objects are then stored in server memory. Each goal object contains its definition and rules that this goal has.

Thus, in the present chapter the main elements of the methodology and the relevant issues have been discussed. The work stages necessary to obtain the desired result of the work have been stated and analyzed. The nature and the major characteristic features of XML as well as JSON advantages over XML have been studied. The attributes of communication between a client and the server have been examined as well as a diagram illustrating this process was shown. Finally, the persistence process has been explained.

3 RESULTS

With the problem solving methods being researched, the actual data models development can start. Data models should include only the relevant conditions, thus making models less verbose and more understandable. Rule conditions should use absract layer which will be convertible to JEXL and not the other way around. Data models should be presented in the common notation such as JSON or XML. Finally, all models should fit into existing software components.

3.1 Used conditions

First, all customer goal specifications were skimmed. As a result, just a few conditions were identified as being constantly repeated. Besides the found conditions, customers named a few that they would like to have as well. Based on those two factors, only five conditions will be supported for now in the editor:

- Current session pageviews: Number of pages user visited in the current browser session.
- Sessions: Number of browser sessions user had since first time on the site.
- Context page views: Number of pages user visited in the goal site section.
- Current session referrer URL: A referrer the user had when entering the site.
- Last page url: The URL of the last page url the user visited.

3.2 Expression model

JEXL is a powerful tool for creating expressions. The language allows creating various sorts of expressions using built-in constructs and the language could also be extended using Java. Such flexibility helps solve complicated problems but, in certain cases, it may as well also cause harm. As JEXL also allows to instantiate any Java class, this can lead to unauthorized access to the database, execution

of malicious scripts, and even crash a JVM. This power would be a problem if JEXL was used in the transmission layer. Everything that comes from the client is untrusted and potentially users could be able to construct arbitrary expressions. For this reason, it was decided to use an abstract layer for describing JEXL conditions. The abstract layer will only allow the use of predefined criteria and operators. An alternative way of solving this problem is described further in the Discussion section of this thesis.

Proceeding to closer analysis, it must be noted that the identified JEXL conditions can be represented in two different formats.

The first format is “criterion operator value”. This format was given an ordered type name. It is easy to notice that any expression that uses order operators (“<” lower than, “>” greater than, “=” equal) can be defined this way.

Ex: “PageViews > 3”.

Instead of algebraic order operators (“<”, “>”, “=”) MongoDB like operators will be used. (Ex: “<” (lower than) is “&lt”). This helps on the server side where Java enum types are used to parse the incoming data model as algebraic operators could not be used in these type definitions.

The second format is “criterion.operator(value)”. This format was called a function type. The function type emulates the class method call and is self-explanatory.

Ex: “CurrentSessionRefererUrl.matches(‘?lang=eng’)”.

3.3 Models notation

At the previous stages of the present research work, the JSON format was chosen to be the best fit for the current task. It works out of a box with JavaScript and will be converted from the string to the object using Gson on the server side.

Gson is a library for serializing and deserializing Java objects into JSON. It will be used because the server has already been using it. Thus, it will help to avoid the unnecessary complexity of introducing a new library to the current implementation.

When a JSON string needs to be deserialized into an object of desired type, Gson uses the type tree of this object type. This helps to instantiate only the expected types and ignores any extra fields (Singh & Letch 2015).

The server will be using the database to persist goal rules. In general, two types of the databases can be distinguished: relational and nonrelational (Hewitt 2011).

Nonrelational databases include object databases, xml databases, document-oriented databases, graph databases and key-value stores, and distributed hash tables. Document-oriented databases, such as MongoDB and others, are a good fit for storing JSON documents as they were specifically built to be used with various document formats: JSON, xml, etc. Moreover, they provide a simple mechanism for storing, searching and fetching the documents.

As for relational databases, they are purposely built for storing structured values in fixed table schemas. There exist solutions that provide a more powerful way for manipulating and persisting document data types. One of the examples is PostgreSQL. This ORDBMS supports managing JSON documents natively, thus allowing document indexing and search.

The current server implementation is using a MySQL database where the goals are stored. Thus a MySQL database will be used and JSON documents will be stored as a string. It is worth noting that this solution will not cause any significant drawbacks as the data will not be searchable and will be retrieved only with the whole row entry.

3.4 Data model on the client side

A single rule consists of a single condition. Every condition contains three parts: criterion, operator, and value. A rule counts as complete only when all condition parts are filled. Each field is dependant on the predecessor field, thus forming a sequence: criterion -> operator -> value. Fields could be filled only in this order. This order was chosen specifically so that the following fields can filter their types and available options based on the parent selection type.

Each criterion field should be unique. However, this condition will be omitted from the data model and will be left to the corresponding programming validation.

The criterion and operator fields are select boxes and value field is a simple text element allowing either number, or string inputs.

```
{
   "dependantOn": "criterion",
   "tagName": "select",
   "options": [
       {
           "name": "$lt",
           "text": "lower than",
           "type": "ordered"
       },
       ...
   ]
}
```

Figure 3. Example of operator field model

Figure 3 shows how the model describes which field should be filled before the operator field. Because the operator field is a select box in the editor, its model also specifies a tag name as “select” corresponding to the html select tag and available options the client can choose from. Before displaying the available options on the screen, they will be filtered by the type of parent value (ex: by “ordered” type). The parent value is determined by looking at the `dependantOn` field value.

```
{
   "dependantOn": "operator",
   "tagName": "input",
   "attributes": {
       "placeholder": "Value"
   },
   "attributesByType": {
       "ordered": {
           "type": "number"
       },
       "function": {
           "type": "text"
       }
   }
}
```

Figure 4. Example of value field model

Figure 4 shows that field can also filter attributes by parent type. Attributes are applied to the actual HTML element defining its behaviour.

Finally, when all fields (criterion, operator and value) are selected and the form is submitted, a rule object is created and sent to the server afterwards.

3.5 Data model in the transmission process

The object which is passed between the server and the client browser is simple. It contains rules relationship describing how rules should be applied:

- either all rules must succeed to trigger a goal;
- or any rule can trigger a goal.

In addition to that, a list of rules is being sent; each rule containing information for each field (criterion, operator, value) and rule name. The rule name is generated automatically as rule, goal name and rule index joined together. The name is used only for debugging.

```
{
   "ruleTie": "allRuleTie",
   "rules": [
       {
           "criterion": "PageViews",
           "operator": "$lt",
           "value": "2",
           "name": "Rule-GoalName-0"
       },
       ...
   ]
}
```

Figure 5. Example of a transmitting object model

Figure 5 shows that the rule object that is being sent contains information only about rule field values without their specifications. By doing so, it minimizes the size of the document.

3.6 Data model on the server side

Once the server has received data in JSON format, it converts the string representation into a rule data transfer object. For each incoming rule field, there is a corresponding Java enum specifying which values are allowed.

```
public enum CriterionType {
   Ordered( "%s %s %s" ),
   Function( "%s.%s(\"%s\")" );
   ...
}
```

Figure 6. Enum defining the allowed criterion types

Figure 6 contains not only the allowed criterion types, but also a string format to convert the Java object into a valid JEXL expression. The string is always filled in the same order: first, comes the criterion, then comes the operator, and last comes the value.

```
public enum RuleCriterion {
   CurrentSessionPageViews( CriterionType.Ordered ),
   Sessions( CriterionType.Ordered ),
   ContextPageViews( CriterionType.Ordered ),
   LastPageEntryMatcher( CriterionType.Function ),
   ...
}
```

Figure 7. Enum defining the allowed rule criteria

Rule criteria specify which type each criterion belongs to.

```
public enum RuleOperator {
   $lt( CriterionType.Ordered, "<" ),
   $gt( CriterionType.Ordered, ">" ),
   $eq( CriterionType.Ordered, "=" ),
   contains( CriterionType.Function, "contains" ),
   matches( CriterionType.Function, "matches" );
   ...
}
```

Figure 8. Enum defining the allowed rule operators

Each rule operator knows about the criterion type it belongs to and how the operator should be presented in the string format.

By using all these enums, a rule could be easily mapped from the JSON representation as it comes to the server (Figure 5) to the Java object. When the rule objects are created, each object has a name, a criterion, an operator, and a value.

The list of converted rules should be validated first as during the transmission process, the data could have been modified by the attacker. While some of these modifications may be harmless, others may be harmful. To protect the server, all rules are validated against the set of allowed conditions. This is done automatically as rules are mapped using enums. Additionally, the criterion and operator fields are type checked. They both should be of the same type either ordered, or function.

After successful data validation, all rule objects are converted into JEXL strings based on their type. A rule object could be easily represented as a JEXL expression simply by printing this object as a formatted string. A string format is specified in the criterion type. These strings override default goal behaviour and are later used to determine when the goal should be triggered. The received JSON string is saved to the database for later use. When the server starts, it fetches rules JSON string from the database, and proceeds with the exact same operation as when the server receives data from the client. Saving goal rules as a JSON string to the database helps when the rules need to be sent to the rules editor. With JSON they could be sent easily with no additional object manipulations.

3.7 Other changes required

Goal rules coming from the rules editor override default goal behaviour. They also need to be persisted in the database. The MySQL database is being used as a storage mechanism for goals. The goal table has been modified to add a new

column where the rules JSON string will be stored. This table manipulation was implemented with a single line SQL expression:

```
ALTER TABLE Goals ADD rulesJSON VARCHAR();
```

Figure 9. SQL expression to add a new column in the Goals table

4 DISCUSSION

The proposed data models fully meet the initial specifications. They are flexible, expandable, use common notation, JSON during transmission and fit in the current server side code.

A new rule condition could be easily added to the existing data models without any need for creating new models. If the condition type did not exist before, it should be added to the server side's allowed types by simply creating another child in `CriterionType` enum. Then the new condition's criterion and operator should be added to the `RuleCriterion` and `RuleOperator` enums respectively. Finally, newly available criteria and operators should be added to the allowed options in the rules editor data model.

4.1 Possible breaking scenarios

Steps to extend data models are minor and not complicated. However, not all changes could be applied to the models when data models operate in the production environment.

By replacing the existing criterion type with another value would break the goal rules already saved in the database as they will not be updated and will refer the unknown types. During type replacement, all existing rules saved in the database should be either removed or modified to refer to the updated type. Type replacement is not a recommended operation. Instead, it is better and simpler to just add a new criterion type.

The rules editor may scale by increasing the number of available criteria and operators. Current data models easily support this type of expanding. This scale may be introduced without massively modifying the data model.

4.2 Alternatives

The JEXL sandbox could be used instead of the proposed data models. Then, the JEXL expressions will be formed right in the rules editor and later sent to the

server. On the server side, the actual sandbox would be implemented. This sandbox would grant user permissions to access and modify the variables they are only allowed to, forbid instantiating any Java classes and allow to use only “safe” and approved methods.

There are no existing JEXL sandbox implementations that are complete and that could be used in the current stack. Because there are not completed attempts (Botha 2011), we would thus, need to implement one by ourselves. The actual sandbox development would take more time than the proposed solution. In addition, the sandbox would be strictly tied to JEXL and would not be flexible Whereas the proposed solution could be used with other expression languages and is not bound to a specific programming language.

5 CONCLUSION

The goal of this thesis was to create a data model for the rules editor. The rules editor is a new software component, thus, it required preliminary research. During this thesis work three data models were created: one for the actual rules editor; a second one for the transmission process and the last one, for the server side.

There was already a change to the data models including the creation of a new rule criterion. The generic process of creation is described in the previous thesis chapter. This creation happened smoothly and without any possible issues, proving predictability and robustness of the implemented data models.

The research results have a defined practical purpose. Currently, the rules editor developed during this thesis work is being used as part of Lekane's software. The project was completed in October 2015, therefore allowing to analyse its impact and consequences. There have been no issues with data models and the rules editor since then. Further improvements may include expanding allowed conditions in the data model and modifying the existing ones. As was proved in this work, these processes are easy and do not require extra development effort. The rules editor is easy to maintain as it does not require any supervision and constant changes.

REFERENCES

Apache Commons (2016). JEXL syntax. Available at: https://commons.apache.org/proper/commons-jexl/reference/syntax.html [Accessed 15 November 2015]

Botha S. 2011. JEXL Secure Sandbox. Available at: http://apache-commons.680414.n4.nabble.com/jexl-JEXL-Secure-Sandbox-td3626959.html [Accessed 23 November 2015]

Bray T., Ed. 2014. The JavaScript Object Notation (JSON) Data Interchange Format. Available at: https://tools.ietf.org/html/rfc7159 [Accessed 29 December 2015]

Fowler M. 2010. Steps Toward The Glory of REST. Available at: http://martinfowler.com/articles/richardsonMaturityModel.html [Accessed 23 January 2016]

Hewitt E. 2011. Cassandra: The Definitive Guide. 1st edn. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Powers C. 2013. Business Rules. Available at: https://github.com/chrisjpowers/business-rules [Accessed 29 October 2015]

Rosenberg J. 2007. The Extensible Markup Language (XML). Available at: https://tools.ietf.org/html/rfc4825 [Accessed 14 December 2015]

Red Hat (2016). Drools – Business Rules Management System. Available at: http://www.drools.org/ [Accessed 25 October 2015]

Red Hat (2016). Hibernate. Available at: http://hibernate.org/ [Accessed 10 January 2016]

Strassner T. (n.d). XML vs JSON. Available at: http://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html [Accessed 17 December 2015]

Service Now (2015). Business Rules. Available at: https://wiki.servicenow.com/index.php?title=Business_Rules#gsc.tab=0 [Accessed 25 October 2015]

Singh I., Letch J. (2015). Gson Design Document. Available at: https://sites.google.com/site/gson/gson-design-document [Accessed 10 January 2016]

Usdin T. B. 2006. How and Why Are Companies Using XML? Mulberry Technologies, Inc. Available at: http://www.mulberrytech.com/papers/HowAndWhyXML/HowAndWhyXML.pdf [Accessed 17 November 2015]